An Introduction to the OS/2 Unicode APIs

By Alex Taylor

Warpstock 2006
Warpstock Europe 2006

Copyright

What You Need

- An OS/2 C compiler and Toolkit.
  - Exercises will include directions for compiling with IBM C Compiler 3.*x* and GCC 3.3.5.
  - GCC 3.3.5 can be provided if necessary.
- At least a basic knowledge of C programming (recommended) or REXX.
  - REXX support is somewhat limited; the extent of coverage will depend on class interest.

What is Unicode?

- Unicode is a character standard designed to allow all possible characters from all known writing systems to be represented on computer systems in an interchangeable form.
  - Defines implementation requirements such as encoding formats, bidirectional algorithms, rendering of composite characters, etc.
  - Assigns every character a unique numeric value between 0 and 1,114,111. This assignment is known as the Universal Character Set (UCS).
- The terms “UCS” and “Unicode” are usually used interchangeably.

Versions of Unicode

- The Unicode standard is maintained by the Unicode Consortium, and is currently up to version 5.0 (released July 2006).
  - Version 1.0 was released in 1991 and supported 65,536 character codepoints (28,302 of which were assigned).
  - Starting with version 2.0, the codespace was expanded to 17 **planes**, each supporting 65,536 codepoints, for a total codespace of 1,114,112 possible characters (but the additional planes were not actually used until version 3.1).
- The Universal Character Set is also published on its own by the International Organization for Standardization as *ISO/IEC 10646*.

Why Unicode?

- Encoding characters by codepage has many disadvantages:
  - Each codepage only supports one or two character sets (usually basic Latin and up to one other) – difficult to support multiple languages simultaneously.
  - Different computer platforms tend to use different codepages, even to represent the same character sets (e.g. IBM 850 vs Windows 1252) – complicates data interchange.
  - DBCS codepages tend to use relatively crude variable-width encoding that makes string parsing difficult.
  - Too many codepages, not enough standardization!

Why Unicode (cont.)?

- Why use Unicode in your application?
  - Access to a wide range of text-manipulation functions which are independent of language or encoding.
  - Multiple character sets can be supported simultaneously, even in a single output stream.
  - Text can be converted to almost any encoding on demand.

UCS Character Values

- Unicode (UCS) character values (or codepoints) are traditionally written as “**U+***”, where * is the character's value in hexadecimal.
- Values U+0000 (0) through U+007F (127) correspond to the standard ASCII character set. It is therefore easy to convert these characters to and from Unicode.
- Values U+0080 (128) through U+00FF (255) correspond to the ISO Latin-1 extensions. This means that the UCS character mapping of U+0000 through U+00FF corresponds exactly to ISO-8859-1 (codepage 819).

The UCS Codespace

- The Universal Character Set encompasses 17 planes, each containing 65,536 codepoints:
  - Plane 0 (U+0000 - U+FFFF): Basic Multilingual Plane (BMP)
  - Plane 1 (U+10000 - U+1FFFF): Supplementary Multilingual Plane (SMP)
  - Plane 2 (U+20000 - U+2FFFF): Supplementary Ideographic Plane (SIP)
  - Plane 14 (U+E0000 – U+EFFFF): Supplementary Special-Purpose Plane (SSP)
  - Planes 15 & 16 (U+F0000 - U+10FFFF): Supplementary Private Use Area
- The BMP contains most characters in use around the world today.

Unicode Character Encoding

- Problem: how can a computer represent each character, consistently and unambiguously, as an integer value between 0 and 1,114,111?
- Traditionally, 1 byte = 1 character. But this only allows 256 values – far too small for the UCS codespace!
- Simplest solution: increase the number of bytes per character.
  - First implemented in UCS-2 encoding.
  - Later expanded to UCS-4 encoding.

UCS-2 Encoding

- UCS-2 is the original Unicode encoding. Introduced back in the days when UCS only defined 65,536 characters (Plane 0).
  - Fixed-width encoding: 2 bytes per character (supporting the value range 0 - 65,535).
  - Advantages: simple, makes data processing easy.
  - Disadvantages: not backwards-compatible with ASCII; only supports Plane 0.
  - Officially considered obsolete, but still used by some implementations.
- (The OS/2 Unicode APIs use UCS-2 internally.)

UCS-4 Encoding

- UCS-4 was introduced to replace UCS-2 when Unicode was expanded to 1,114,112 characters.
  - Fixed-width encoding: 4 bytes per character, supporting the entire range of UCS values (and then some).
  - Advantages: simple, makes data processing easy; supports entire UCS codespace.
  - Disadvantages: requires a huge amount of storage; not backwards-compatible with ASCII, or even with UCS-2.
  - Rarely used.
- Nowadays, UCS-4 is more commonly called “UTF-32”.

Problems with Fixed-Width Encoding

- Once the UCS codespace was expanded to 17 planes, fixed-width character encoding became impractical.
  - Four bytes per character is wasteful, especially since (in practice) most characters fall within Plane 0 (BMP). (See example.)
  - Most Unicode libraries were originally designed to use two-byte characters (UCS-2); redesigning them for four-byte characters (UCS-4) can break compatibility.
  - Fixed-width multi-byte encodings are not compatible with ASCII data streams (which are widely used).
  - UCS-4 isn't even compatible with UCS-2 data streams!
- A better solution: variable-width encoding.

UTF-16 Encoding

- Intended to seamlessly replace UCS-2.
  - Uses 2 bytes per character to encode characters within the BMP (U+0000 - U+FFFF).
  - Uses 4 bytes per character to encode characters outside the BMP (U+1FFFF and up). Leading bytes of 4-byte characters always fall within the range U+D800 - U+DFFF, which are reserved within the BMP and so cannot start legal 2-byte characters.
  - Any legal UCS-2 character is also a legal UTF-16 character.
  - Advantages: backwards-compatible with UCS-2; requires no more than 2 bytes to represent any BMP character; supports the entire UCS codespace.
  - Disadvantages: slightly more complex than UCS-2; still not backwards-compatible with ASCII data streams.

UTF-8 Encoding

- Possibly the most ingenious encoding of all.
  - Uses between 1 and 4 bytes for each character:
    - 1 byte for values U+0000 - U+007F (basic ASCII)
    - 2 bytes for values U+0080 - U+07FF
    - 3 bytes for values U+0800 - U+FFFF
    - 4 bytes for values U+10000 and up.
  - A 1-byte character always starts with a leading 0 bit.
  - The first byte of a multi-byte character always starts with a number of leading 1 bits equal to the number of bytes used (i.e. 110*xxxxx*, 1110*xxxx* or 11110*xxx*). Following bytes always start with 10. The remaining bits are used to encode the character value.
  - All ASCII text (i.e. characters below U+0080) is also valid UTF-8 text!

UTF-8 Encoding (cont.)

- Advantages of UTF-8:
  - Backwards-compatible with basic ASCII.
  - Supports the entire UCS codespace.
  - Requires much less storage than UCS-4, and (on average) slightly less than UCS-2.
  - Every single byte in a data stream is instantly identifiable as a leading byte, a following byte, or a single-byte character (which helps prevent data corruption).
- Disadvantages:
  - Relatively complex algorithm, resulting in higher processing overhead.
- UTF-8 is typically used for text output and interchange, not for internal processing.

Unicode Under OS/2

- The OS/2 implementation of Unicode uses UCS-2 for internal processing.
- Characters outside the BMP are not supported (AFAIK).
- OS/2 supports three Unicode encodings: UCS-2, UTF-8, and UPF-8 (an OS/2-specific proprietary implementation used for output under PM).
- The OS/2 Unicode API (Universal Language Support) is available under Warp 4 and up, or Warp 3 with a recent FixPak. (Some versions of the Java 1.1.8 runtime package will also install or update it as needed.)

The Unicode API

- The OS/2 Unicode functions (a.k.a. “Universal Language Support”) cover four major categories:
  - *Unicode Text Processing*: handling Unicode (UCS-2) text.
  - *Conversion*: converting text to and from Unicode, or from one codepage to another.
  - *Localization*: presenting information according to national or cultural conventions.
  - *Keyboard Input*: converting keyboard input to and from Unicode according to different keyboard layouts.

Using the Unicode API

- Header files:
  - **unidef.h**: main API definitions (text processing & localization)
  - **uconv.h**: conversion API definitions
  - **unikbd.h**: keyboard API definitions
- Import libraries (when using the IBM Toolkit):
  - **libuls.lib**: text processing & localization functions
  - **libconv.lib**: conversion functions
  - **unikbd.lib**: keyboard functions

Representing Unicode Text: UniChar

- All Unicode text is encoded internally as UCS-2.
- A Unicode character is represented by the **UniChar** data type (a two-byte integer value):

```
typedef unsigned short UniChar;
```

- A UniChar is also sometimes referred to as a “UCS code element”.

Setting UniChar Values

- Directly assign the character's UCS value as an integer.

```
UniChar uc = 0x0041;    // uppercase 'A' = U+0041
```

- Use C wide-character conventions (assuming **wchar_t** is an unsigned short):

```
UniChar uc = L'A';
UniChar *puniStr = (UniChar *) L"Welcome to
  Warpstock";
printf("%lc\n", uc );
printf("%ls\n", puniStr );
```

Only use this technique when dealing with basic ASCII characters!

Setting UniChar Values (cont.)

- Use the codepage conversion functions:
  - *UniStrToUcs()* and *UniUconvToUcs()* will convert a multibyte string from any known codepage into a UniChar string.
  - *UniStrFromUcs()* and *UniUconvFromUcs()* will convert a UniChar string into a multibyte string in the desired codepage.

Working with UniChar Strings

- Take the string:

```
Hello world
```

- Under a normal (ASCII-based) multibyte codepage, this string is represented by:

```
48 65 6C 6C 6F 20 77 6F 72 6C 64 00
```

- As a UniChar string, it becomes:

```
00 48 00 65 00 6C 00 6C 00 6F 00 20 00 77 00 6F 00 72 00 6C
00 64 00 00
```

Working with UniChar Strings (cont.)

- Most standard string-handling functions treat a zero byte as a string termination character, and therefore will not handle this string correctly.
- As mentioned, C wide-character APIs can sometimes be used (where available). But:
  - In practice, this is only possible with basic ASCII text (since these APIs cannot convert the Unicode values into meaningful characters for the current codepage).
  - Also, there is no way to use wide-character conventions in conjunction with (for instance) PM controls.
- Consequently, ULS provides its own functions for handling UniChar text.

Unicode Input and Output

- In general, UniChar strings must be converted to another format before output.
  - Convert to a Unicode encoding which is designed for output:
    - UPF-8 (codepage 1207) - preferred for use under Presentation Manager.
    - UTF-8 (codepage 1208) - preferred for data interchange (files, e-mail, etc.).

    This does not allow direct display in OS/2 text sessions.
  - Convert to another codepage. This allows display of text in OS/2 window or full-screen sessions by converting to the current process codepage.
- Conversely, input routines can accept text in the current codepage, and then convert it to UCS-2 for subsequent processing.

Unicode String Manipulation

- Most standard library functions cannot be used with UniChar text (due to embedded zero bytes). Therefore, ULS includes Unicode-enabled equivalents of several standard C library functions:
  - The *str*()* functions of the standard string library.
  - The character transformation functions *tolower()* and *toupper()*.
- In addition, several functions are provided for querying and transforming UniChar characters and strings in a locale-dependent way.

Exercise 1:
Using UniChar Strings

- Exercise:
  - Review the sample program. Make sure you understand what it does.
  - Verify that the program compiles and runs successfully.
- Objectives:
  - Verify compiler and toolkit setup.
  - Verify program using ULS API can be compiled and run.
  - A brief introduction to UniChar string manipulation.

Codepage Conversion

- The ULS conversion functions allow text to be converted from any multibyte codepage into the equivalent strings in UCS-2 (UniChar) format, and vice versa.
- Text can also be converted from one multibyte codepage to another by going through UCS-2 as an intermediate step.
- Logical character value (not byte value) is preserved across codepages, whenever possible.
- If a character does not exist in the target codepage, it is replaced by a designated "substitution character".

UconvObject

- The **UconvObject** type is used to control conversions (in either direction) between UCS-2 and one particular codepage.
- Use *UniCreateUconvObject()* to create a UconvObject for the specified codepage:

```
UconvObject uconv;
UniCreateUconvObject( (UniChar *) L"IBM-850", &uconv );
```

- Use *UniFreeUconvObject()* when the UconvObject is no longer required:

```
UniFreeUconvObject( uconv );
```

Conversion Specifiers

- The first parameter to *UniCreateUconvObject()* is a UniChar string called the conversion specifier: "***<codepage-name>*[@][*<modifers>*]**"
- The codepage name must be a legal OS/2 codepage identifer, normally in the form "IBM-*x*" where *x* is the codepage number. Certain aliases are also defined (in `\LANGUAGE\CODEPAGE\UCSTBL.LST`), and some constants are provided in `uconv.h`.
- *UniMapCpToUcsCp()* will generate a legal codepage name from the specified codepage number.

Conversion Modifiers

- The optional modifers define certain attributes of the conversion object (separated by commas).
  - **map**: Defines how control bytes should be treated during conversion. *Default: “map=data”*
  - **path**: Indicates whether strings are assumed to contain paths (DBCS only). *Default: “path=yes”*
  - **endian**: Indicates the UCS-2 byte order (endian) to use. *Default: “endian=system”*
  - **sub**: Indicates whether character substitution is enabled. *Default: “sub=from-ucs”*
  - **subchar**: Indicates the substitution character to use in codepage strings. *(Default varies by codepage.)*
  - **subuni**: Indicates the substitution character to use in UCS-2 strings. *Default: “subuni=\xFFFD”*

Performing Conversions

- Once you have a UconvObject, you can two different sets of functions to perform the conversion(s).
  - *UniStrToUcs()* and *UniStrFromUcs()*
  - *UniUconvToUcs()* and *UniUconvFromUcs()*
- The *UniStr*()* functions are easier to use; however, the *UniUconv*()* functions allow for slightly more flexible error-checking and recovery.
- The *UniStr*()* functions were not available in the first versions of the Unicode API; they were added in an early Warp 4 FixPak (and a late Warp 3 FixPak).

UniStrToUcs / UniStrFromUcs

- *UniStrToUcs()* and *UniStrFromUcs()* use a fairly simple syntax.
- Conversion is atomic: it either succeeds or it fails (reflected in the return code).
- Substitution is always performed (regardless of the conversion modifers used to create the UconvObject).
- Parameters:
  - UconvObject
  - Output buffer (must be already allocated)
  - Input string
  - Length of output buffer (in characters)

UniUconvToUcs / UniUconvFromUcs

- *UniUconvToUcs()* and *UniUconvFromUcs()* use a fairly complex syntax.
- An error may cause conversion to stop part-way through, and more information is available to allow error recovery.
- Parameters:
  - UconvObject
  - Pointer to input string (must already be allocated)
  - Pointer to input string length (in characters)
  - Pointer to output buffer (must already be allocated)
  - Pointer to output buffer length (in characters)
  - Pointer to number of substitutions made

Output Buffer Length

- The output buffer must be large enough to contain all converted characters, plus NULL.
  - When converting to UCS-2, the UniChar output buffer will never be longer (in UniChars) than the length of the input buffer (in bytes), not including the terminating NULL.
  - When converting from UCS-2, the output string **may be longer** than the UniChar input string, if the target codepage allows 2-, 3-, or 4-byte characters.
    - If you know the target codepage contains only single-byte characters, the output buffer length may be the same as the input length (not including the terminating NULL).
    - If the character width of the target codepage is undetermined, you should allocate at least 4 output bytes per input UniChar, because OS/2 MBCS codepages may represent a single character using up to 4 bytes.

Character Substitution

- If the input string contains characters that do not exist under the target codepage, the character is (normally) replaced in the output string by a generic “substitution character”.
- Every codepage has its own default substitution character (which may be changed through the UconvObject attributes). It should always be a displayable glyph under the target codepage.
- If substitution is disabled (through the UconvObject attributes), an error condition will be returned whenever an unsupported character is encountered. (Applies to the *UniUconv** functions only.)

Exercise 2: Codepage Conversion

- Exercise:
  - Review the example program(s).
  - Modify the example program to convert the input text to UTF-8 instead of HTML.
- Objectives:
  - Demonstrate the conversion process.
  - Illustrate the difference between the *UniStr** and *UniUconv** functions.

Localization

- Localization is based on the concept of a *locale*: a set of conventions associated with a particular language or culture that specifies how information should be presented.
- These conventions include such things as:
  - The names of the country and language normally associated with the locale.
  - The default currency unit.
  - The standard number format.
  - The standard time and date format.
  - The standard units of measurement.
  - The default codepage(s) associated with the locale.
  - Rules for text classification and transformation.

How Locales Work

- Locales are used for:
  - Determining character types or text transformation rules when using certain Unicode functions.
  - Modifying application behaviour as appropriate for the current environment.
- There are two types of locale:
  - *System locales*: standard locale constants defined by OS/2. They cannot be modified or deleted. System locales exist for every country and language known to OS/2, and are used to identify standardized conventions.
  - *User locales*: customizable locale instances which may be created, modified, and/or deleted. User locales are derived from system locales, but represent the user's own preferences rather than generic standards.

Identifying Locales

- A **LocaleObject** is used as a handle to a specific locale. A variable of this type is passed to certain functions in order to identify the locale being used.
- *UniCreateLocaleObject()* is used to initialize a LocaleObject for the specified locale.
  - The locale is usually identified by UCS-2 name (“en_US”, “de_DE_EURO”, etc.).
  - Specifying an empty string will create a locale based on the current environment settings (e.g %LANG%).
- A LocaleObject should be freed using *UniFreeLocaleObject()* once it is no longer needed.

Identifying Locales (cont.)

- *UniQueryLocaleList()* can be used to query the names of all existing locales (system, user, or both).
- *UniMapCtryToLocale()* returns the locale name corresponding to the specified country code.
- *UniQueryLocaleObject()* can be used to obtain the locale name associated with an existing LocaleObject.

Getting Locale Information

- Every locale consists of a fixed number of key-value pairs, known as *locale items*, which describe the conventions for that locale.
- Each locale item is referred to by a key constant. The prefix indicates the data type:
  - **LOCI_s**, **LOCI_j**, and **LOCI_w** indicate string values.
  - **LOCI_i** and **LOCI_x** indicate integer values.
- To query the value of a specified locale item:
  - *UniQueryLocaleItem()* writes the current value of any locale item to a string variable.
  - *UniQueryLocaleValue()* writes the current value of an integer locale item to an integer variable.

The Locale Conventions Structure

- The locale conventions structure (**struct UniLconv**) contains information about how numbers and currency values are represented by a locale.
- It is designed to be analogous to the C **lconv** structure (returned by the *localeconv()* library function).
- This structure may be obtained for a specified locale using *UniQueryLocaleInfo()*.

Working With User Locales

- User locales are used to represent locally-configured preferences.
- Normally, user locales are created and modified by the user through the OS/2 Locale applet. However, it can also be done programmatically.
  - *UniMakeUserLocale()* creates a new user locale, by copying an existing locale (system or user).
  - *UniDeleteUserLocale()* deletes a user locale.
  - *UniSetUserLocaleItem()* modifies individual items within a user locale.
- Changes do not take effect outside the current process until *UniCompleteUserLocale()* is called. This function writes all user locales to disk.

Exercise 3: Getting Locale Information

- Exercise:
  - Review the example program.
  - Modify this program to display selected items from each locale (e.g. the associated country and language names).
- Objective:
  - Demonstrate how to identify and query locales.

Character Attributes

- A character *attribute* (or *classification*) describes categories to which a character belongs.
- Attributes include standard POSIX types like “alpha” and “digit”, but also extended Unicode types like “ideograph” and “nonspacing”.
- There are also attributes for describing text layout, and others indicating the specific Unicode character sets to which a character belongs.

Attribute Names and Identifiers

- Every attribute has a integer *identifier* (referenced by symbolic constant).
- Most (though not all) attributes also have a *name*, which is a human-readable UniChar string.
- *UniQueryAttr()* may be used to obtain the identifier value associated with an attribute name.

Attribute Categories

- Character attributes are grouped into several different categories:
  - Character type attributes (names start with a lowercase letter):
    - POSIX types (identifier symbols start with **CT_**)
    - Extended types (identifier symbols start with **C3_**)
    - Win32 compatibility types (identifier symbols start with **C1_**; no names)
  - Character set attributes (names start with “_”)
  - Layout/BIDI attributes (names start with “**#**”)

Localized Attributes

- An **AttrObject** is used to represent attributes in a locale-specific context.
  - *UniCreateAttrObject()* creates an AttrObject for the specified attribute(s) and locale.
  - *UniFreeAttrObject()* frees an AttrObject when it is no longer needed.
  - *UniQueryCharAttr()* queries a UCS-2 character's attributes using an AttrObject.
  - *UniScanForAttr()* searches a UniChar string for characters matching the specified AttrObject.
  - Standard POSIX attributes may be queried using various *UniQuery** functions (similar to the C “ctype” functions).

Locale-Independent Attribute Functions

- There are several functions which may be used query attributes independently of locale.
  - *UniQueryChar()* is used to determine whether a single UCS-2 character has the specified POSIX or Win32 character-type attributes (CT_* or C1_*)
  - *UniQueryStringType()* returns an array of integer bitmasks for a UniChar string, where each bitmask describes the attributes (in the requested category) of a single character.
  - *UniQueryCharType()* returns all the attributes of a single UCS-2 character in a **UNICTYPE** data structure.

Transforming Text

- ULS provides various functions for transforming UniChar strings in locale-specific ways.
  - *UniTransLower()* converts text to lowercase.
  - *UniTransUpper()* converts text to uppercase.
  - *UniTransformStr()* performs advanced string transformations.
- Note that the input and output strings are not guaranteed to be the same length. Make sure you provide a large enough buffer!

XformObjects

- An **XformObject** is used to define a transformation, for use with *UniTransformStr()*. It has two attributes:
  - The transformation type
  - The locale being used
- Use *UniCreateTransformObject()* to create an XformObject.
- Use *UniFreeTransformObject()* to dispose of it when done.

Transformation Types

- The following transformation types are valid for all locales:
  - **lower**: Convert text to lowercase.
  - **upper**: Convert text to uppercase.
  - **compose**: Convert character-diacritic combinations into single glyph forms.
  - **decompose**: Convert diacritical forms into separate characters and diacritics (combining marks).
  - **hiragana**: Convert Japanese phonetic characters into Hiragana.
  - **katakana**: Convert Japanese phonetic characters into Katakana.
  - **kana**: Convert Japanese phonetic characters into half-width Katakana.

Exercise 4: Transforming Text

- Exercise:
  - Review the example program.
  - Modify this program to remove all accents from the string (i.e. convert accented characters into non-accented ASCII characters).
- Objective:
  - Demonstrate Unicode text transformations.

Additional Information

- OS/2 Toolkit documentation is obsolete and/or incomplete. Project to provide updated documentation:

  http://www.cs-club.org/~alex/os2/toolkits/uls/index.html

- General Unicode information is available from the Unicode Consortium:

  http://www.unicode.org/

- Useful information & samples from one of the OS/2 Internationalization developers:

  http://www.borgendale.com/uls.htm

Questions?